**Поток –** абстракция, которая используется для чтения и записи информации. Упорядоченная последовательность данных, которым соответствует определенный источник или получатель.

Все потоки ведут себя одинаково, несмотря на отличие в физических устройствах. То есть взаимодействие как с файлом, так и с консолью (и вообще чем угодно) будет одинаковым.

Потоки можно использовать для

* Чтения и записи файлов
* Поток может быть связан с сетевым сокетом, с помощью которого можно отправлять и получать данные по сети.
* Чтение и вывод в консоль.

Потоки делятся на 2 типа:

* **Поток ввода –** поток, с помощью которого можно прочитать данные.
* **Поток вывода –** поток, с помощью которого можно записать данные.

Также они разделяются на:

* **Символьные (Используют Unicode)**
* **Байтовые**

В Java потоки реализованы в пакете java.io. В нем определены четыре абстрактных класса для работы с потоками:

* **InputStream** – байтовый поток ввода
* **OutputStream** – байтовый поток вывода
* **Reader** – символьный поток ввода
* **Writer** – символьный поток вывода.

**FileInputStream и FileOutputStream**

Класс **FileOutputStream** предназначен для записи байтов в файл и унаследован от OutputStream.

В конструктор можно передать либо путь к файлу в виде строки, либо объект File. Вторым параметром *append* можно указать, дописывать ли данные в конец файла, либо перезаписать его.

Если такого файла нет, он автоматически создастся при записи.

Для записи используется метод write(byte[] data);

Класс **FileInputStream** предназначен для считывания данных из файла и является наследником InputStream.

В конструктор можно передать путь к файлу. Если файла нет, либо он не может быть открыт – генерируется FileNotFoundException.

Для чтения используется метод read() который возвращает следующий байт. При достижении конца файла возвращается -1. Можно читать также в массив байт.

Оба класса предназначены для чтения и записи двоичных файлов.

**FileReader и FileWriter** используются для чтения и записи в файл символьных данных.

**ByteArrayInputStream и ByteArrayOutputStream**

Используются для работы с массивами байтов.

**ByteArrayInputStream** используется для чтение данных из массива байтов. В конструктор принимает массив байтов (опционально смещение и кол-во символов).

**ByteArrayOutputStream** используется для вывода данных в массив байтов. Может принимать в конструктор размер выходного массива (по умолчанию 32 байта).

Оба потока не требуют закрытия явным образом.

**ФИЛЬТРУЕМЫЕ ПОТОКИ**

Они представляют собой оболочки, в которые заключаются базовые потоки ввода вывода для расширения их функциональных возможностей. Типичными операциями являются буферизация, преобразование символов и исходных данных. Фильтруемые байтовые потоки реализуются в классах **FilterInputStream** и **FilterOutputStream.**

**BufferedInputStream и BufferedOutputStream**

Данные классы наследуются от Фильтруемых потоков, используются для оптимизации операций ввода вывода. Они имеют специальный буфер в памяти. Они оборачивают собой обычные потоки.

**BufferedInputStream** накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода. Имеет 2 конструктора:



Так же имеет метод *mark()* чтобы запомнить место в потоке ввода, чтобы в дальнейшем вернуться в это место с помощью *reset()*.

**BufferedOutputStream** создает буфер для потоков вывода. Накапливает вводимые данные, и когда буфер заполнен, производится запись данных. Конструктор аналогичен BufferedInputStream. Содержит метод *flush()* чтобы принудительно записать данные из буфера.

Аналогично работают классы **BufferedReader** и **BufferedWriter**. BufferedReader имеет метод *readLine*, который считывает строку целиком и возвращает String(). Есть так же метод *lines()* возвращающий Stream из строк файла.

**PushbackInputStream** реализует механизм возврата в поток. Имеет метод *unread()* который может вернуть обратно в поток байт, либо массив байт.

Аналогично действует **PushbackReader**.

**DataOutputStream и** **DatainputStream**

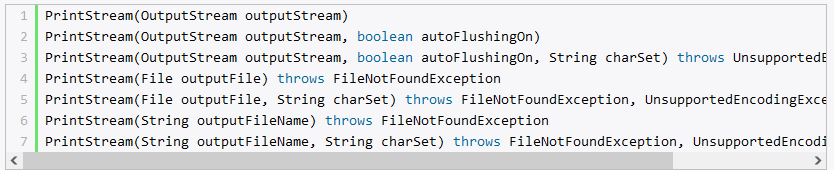
Классы, позволяющие выводить примитивные данные в поток или вводить их из потока. Они преобразуют примитивные типы в последовательности байтов и наоборот. В DataOutputStream:

 Аналогичные методы (*read*) есть и в DatainputStream.

**PrintStream и PrintWriter**

Класс **PrintStream** используется для вывода на консоль любых объектов в Java используя метод *toString()*. System.out.println() как раз использует его (out – объект класса PrintStream, а println() его метод).

Этот класс можно использовать и для записи в поток вывода, передав его в конструктор. Можно также передать объект File либо путь к файлу.



В него также добавлено автоочистка при при каждой записи \n, записи массива байтов или вызове println().

Основные методы PrintStream:

* println(): вывод строковой информации с переводом строки
* print(): вывод строковой информации без перевода строки
* printf(): форматированный вывод

**PrintWriter** можно использовать как для вывода информации на консоль, так и в файл или любой другой поток вывода. Используется для записи символьной информации. Упрощает интернационализацию программ (за счет Unicode).

**SequenceinputStream** позволяет соединить вместе несколько потоков типа InputStream. В процессе работы он по очереди считывает потоки до их исчерпания.

**Закрытие потоков**

После завершения потоки нужно закрывать методом **close()** который определен в интерфейсе Closeable. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Лучше всего закрывать потоки с помощью try-with-resources.

**Класс File**

Оперирует непосредственно файлами и каталогами. В нем не определяется каким образом читаются и пишутся данные, но описываются свойства самих файлов (Права доступа, размер, время, дата и путь к каталогу, перемещение по иерархиям подкаталогов а также манипулирования этой информацией).

С помощью данного класса можно переименовывать и удалять файлы и каталоги. В каталоге можно также создавать подкаталоги.

**Flushable**

Данный интерфейс имеет единственный метод flush(). Этот метод вызывается чтобы принудительно отправить буферизированные данные в поток вывода (записать их на физическое устройство).

**СЕРИАЛИЗАЦИЯ**

**Сериализация** – процесс сохранения состояния объекта в последовательность байт. Оказывается удобной, если нужно сохранить состояние программы в таком

**Десериализация –** это процесс восстановления объекта из последовательность байт.

Сериализация используется для реализации удаленного вызова процедур (Remote Method Invocation).

Средствами сериализации может быть сохранен и восстановлен только объект класса, реализующего интерфейс *Serializable*. Это маркерный интерфейс. Если класс сериализуется, то сериализуются и все его подклассы.

Чтобы переменная не сериализовывалась, нужно объявить ее с ключевым словом **transient**.

Интерфейс **ObjectOutput** поддерживает сериализацию объектов. В нем объявляется метод *writeObject()*, который вызывается для сериализации объекта.

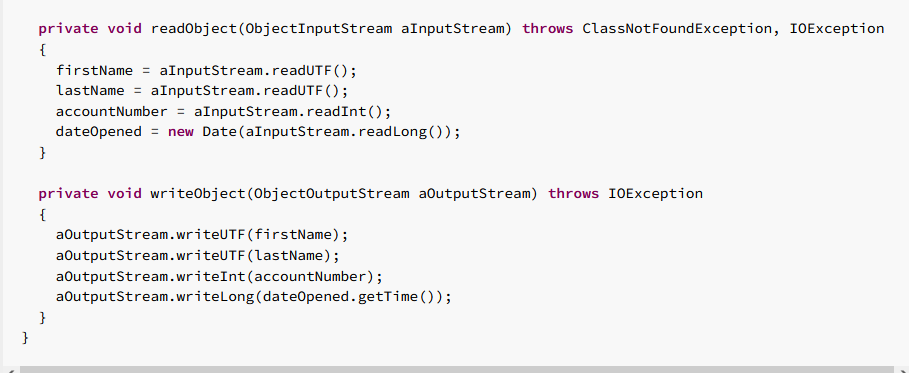
Класс **ObjectOutputStream** отвечает за вывод объекта в поток.



В интерфейсе **ObjectInput** определяется метод *ReadObject()*, который вызывается для десериализации объектов.

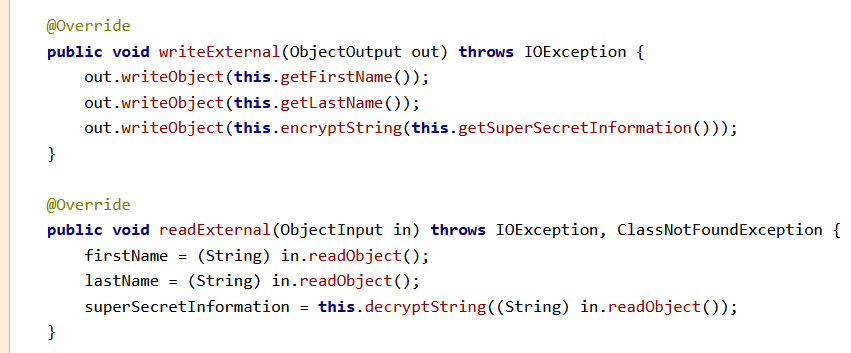
Класс **ObjectInputStream** отвечает за ввод объекта из потока.

Для реализации кастомной сериализации, нужно заимплементить интерфейс *Serializable* и в классе определить 2 метода: *readObject*() и *writeObject*(). Эти методы должны принимать объекты классов *ObjectInputStream* и *ObjectOutputStream.* С помощью методов readXXX и writeXXX описать сериализацию.



Если нужно выполнить какую-то логику перед сериализацией или после десериализации, и при этом стандартный механизм сериализации нас устраивает, можно использовать методы *defaultReadObject*() и *defaultWriteObject*() внутри тех же readObject() и writeObject().

Еще один способ описания кастомной сериализации имплементация **Externalizable.** При этом нужно реализовать 2 обязательных метода: writeExternal(ObjectOutput out) и readExternal(ObjectInput in).



При использовании Externalizable у класса должен быть конструктор по умолчанию. Ведь прежде чем вызвать метод readExtenal, должен быть создан объект класса.

У каждого класса, реализующего Serializable должен быть уникальный идентификатор версии класса. По умолчанию генерируется компилятором на основании полей и методов. И если мы что-то поменяем в классе, и значения *serialVersionUID* не совпадают при десериализации, будет выброшено исключение *InvalidClassException*.